FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Embedded scheduler for dynamically reconfigurable accelerators

Carlos Jorge Matos Carneiro de Sousa

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Paulo de Castro Canas Ferreira

Co-supervisor: Nuno Miguel Cardanha Paulino

June 2016

Resumo

Atualmente, o volume de dados e a complexidade das aplicações tem crescido de forma exponencial. Simultaneamente, os sistemas embarcados estão-se a tornar cada vez mais amplamente utilizados. Para se continuar a responder à crescente necessidade de poder computacional, é necessário projetar circuitos mais rápidos e eficientes em termos de energia. No entanto, como o tamanho diminui e a densidade de circuitos aumenta, as soluções atuais estão rapidamente a atingir a sua capacidade máxima. Com isto em mente, é necessário desenvolver novas arquiteturas de computadores. Uma abordagem possível passa por sistemas dinamicamente reconfiguráveis. A adição de hardware configurável irá permitir a optimização de pequenas partes da execução da aplicação, conduzindo a um aumento geral da velocidade do sistema. O principal objetivo deste projeto de dissertação é implementar um escalonador embarcado capaz de, durante a execução, gerar as configurações e o escalonamento de instruções para uma Unidade de Hardware reconfigurável, responsável pela aceleração partes da aplicação mais frequentemente executadas e identificadas a partir de rastros de execução binários.

Abstract

Nowadays, the volume of data and application complexity is growing sharply. Simultaneously, embedded systems are becoming more widely used. To keep up with the increasing need of computational power, it is essential to design faster and more power efficient circuits. However, as size decreases and the density of circuits increases, the present day solutions are steadily hitting their maximum capability. With this in mind, it is necessary to develop new computer architectures. One approach is to have a dynamically reconfigurable system. The existence of configurable hardware would allow the optimization of small parts of the application's execution, leading to an overall speedup of the system. The main goal of this project is to implement an embedded scheduler capable of, during runtime, generating the specification and the operation schedule of a Reconfigurable Processing Unit that accelerates the execution of application hot spots identified from binary execution traces.

Acknowledgments

I would like to thank my supervisor, João Paulo de Castro Canas Ferreira, for his guidance and insight and Nuno Paulino and Mário Ferreira for all the help they provided. I am grateful to my colleagues for their support during the development of this project. Lastly, I thank my parents and my girlfriend for an entirely different genre of support altogether.

Carlos Sousa

"We explore because we are human, and we long to know."

Stephen Hawking

Contents

List of Figures

List of Tables

Abbreviations

ALAP	As Late As Possible
ASAP	As Soon As Possible
CDFG	Control and Data Flow Graph
CGRA	Coarse-Grained Reconfigurable Array
Estart	Earliest Start Time
FPGA	Field Programmable Gate Array
FU	Functional Unit
GPP	General Purpose Processor
II	Iteration Interval
MB	Megablock
minII	Minimum Iteration Interval
recII	Recurrence Iteration Interval
resII	Resource Iteration Interval
RPU	Reconfigurable Processing Unit

Chapter 1

Introduction

1.1 Motivation

Nowadays, the volume of data and application complexity is growing sharply. Simultaneously, embedded systems are becoming more widely used. To keep up with the increasing need of computational power, it is essential to design faster and more power efficient circuits. However, as size decreases and the density of circuits increases, the present day solutions are steadily hitting their maximum capability. With this in mind, it is necessary to develop new computer architectures.

1.2 Contextualisation

The idea of utilizing configurable hardware to accelerate the execution of programs or part of them is, nowadays, a subject of study by many researchers.

One approach is to have a dynamically reconfigurable system. The existence of configurable hardware would allow the optimization of small parts of the application's execution, leading to an overall speedup of the system. The biggest drawback here is the effort spent to manually design custom hardware, like ASICs, which is a very time-consuming and hard to scale task.

Other approach is to accelerate applications running on a general purpose processor (GPP), by translating some parts of their execution to reconfigurable hardware, a Reconfigurable Processing Unit (RPU). This is the case of [1].

The system presented in [1] is the base of the work developed in this dissertation. The approach can de divided in smaller steps:

- Designing one or more RPU specifications
- Identification of repeating patterns of executed instructions (Megablocks)
- Translation of the Megablocks to a RPU specification
- Modifying the application to use the RPU

The system is capable of, in runtime, transfering the execution from the GPP to the RPU, but the selection and organization of RPU resources are made offline. In an early stage, an embedded scheduler should be able to map the megablocks to the existing RPU. So, all the work related with the translation of "the GPP instructions to architecture independent operations" [2], the design of CDFG, the execution of the modulo scheduling and the translation to the existing RPU should be adapted to be made online. This way, it would be possible to achieve an autonomous and self accelerating system.

1.3 Objectives

The main goal of this work was to develop an embedded software-based scheduler capable of mapping (previously-detected) Megablock operations and selecting the best accelerator to use (from a set of pre-defined accelerators) on the RPU.

This being said, it was important to define a set of objectives that would be a guideline for the work progress throughout the development of the system:

1. Design and implementation of the embedded software scheduler (1st version),
2. Adaptation of the implementation to generate the configuration words for the RPU's,
3. Validation and verification,
4. Improved version of the scheduler.

The first objective was the one that was more time consuming. It included the study of different operation scheduling techniques and the implementation of the chosen algorithm. The selected method should be capable of selecting and organizing the RPU resources depending on the Megablock that is going to be executed. In order to achieve this, it was necessary to translate the instructions of each one to a set of operations that are available on the RPU.

Since the identification of Megablocks is still made offline, it was not necessary to modify the structure of the available system. The reconfiguration of data paths, involving the RPU and the data memory that is shared with the GPP is made by generating the same configuration words of the offline scheduler. These configurations are also responsible for the connections between the output and input registers of the RPU, since the scheduling algorithm will explore instruction-level parallelism.

Validation and verification is a natural objective of any project, and it aims to check the proper functioning of the system. The correctness of the implementation is proven in 5.

Finally, after the completion of all the other objectives, it was interesting to improve the developed system, in order to allow the addition of features, such as choosing between a set of RPU's.

1.4 Thesis Organization

In this section, a small contextualization of the theme under study has been presented. Following are 5 other chapters. Chapter 2 details the state of the art, which contains a description of available systems and a number of scheduling techniques. In chapter 3 the problem in study is described and the objectives are presented. In chapter 4 the currently implemented system is detailed. Chapter 5 presents the obtained results throughout this work. Finally, chapter 6 contains a small description of possible future modifications of the implemented system and some conclusions about the work developed.

Chapter 2

Literature Review

This chapter presents some relevant state-of-art information related to available systems and software and some operation scheduling techniques.

2.1 Analysis of Available System and Software

The system implemented in [1] is the base of the entire developed.

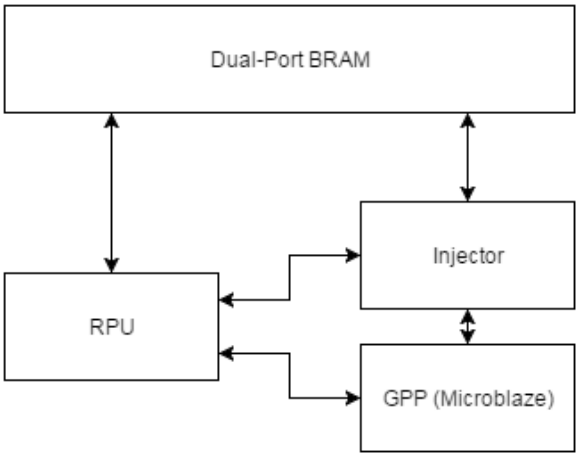

Figure 2.1: Simplified system architecture of the system developed in [1]

Figure 2.1 represents a simplified architecture of the whole system. The MicroBlaze General Purpose Processor (GPP) has no alteration but there is an accelerator coupled as a co-processor, which is the Reconfigurable Processor Unit (RPU). Both processors have access to the dual-port Block RAM (BRAM) that stores code and data. The migration of the execution from the GPP to the RPU is controlled by the Injector module, which is capable of identifying the imminent execution of the previously detected Megablocks.

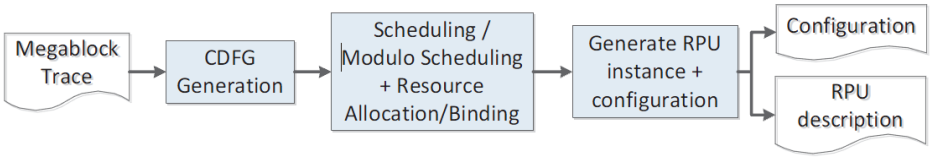

Figure 2.2: Stages of the RPU generation of the system developed in [1]

Repeating patterns of executed instructions, which are named Megablocks in [1], are the focus of the acceleration mecanism. The Megablocks, which are associated with loop behaviour, are identified from execution traces. Figure 2.2 illustrates the main Megablock mapping and RPU generation stages. The Megablocks are then translated into a Control and Dataflow Graph (CDFG), Figure 2.3. This CDFG is the input of the scheduler, which is responsible for generating the RPU instance and the system configuration. The RPU can be customized for each Megablock, modifying the number of Functional Units (FU)'s and the connections between the FU's. After scheduling a Megablock, the system is responsible for the generation of interconnections that allow the translation of the execution to the RPU.

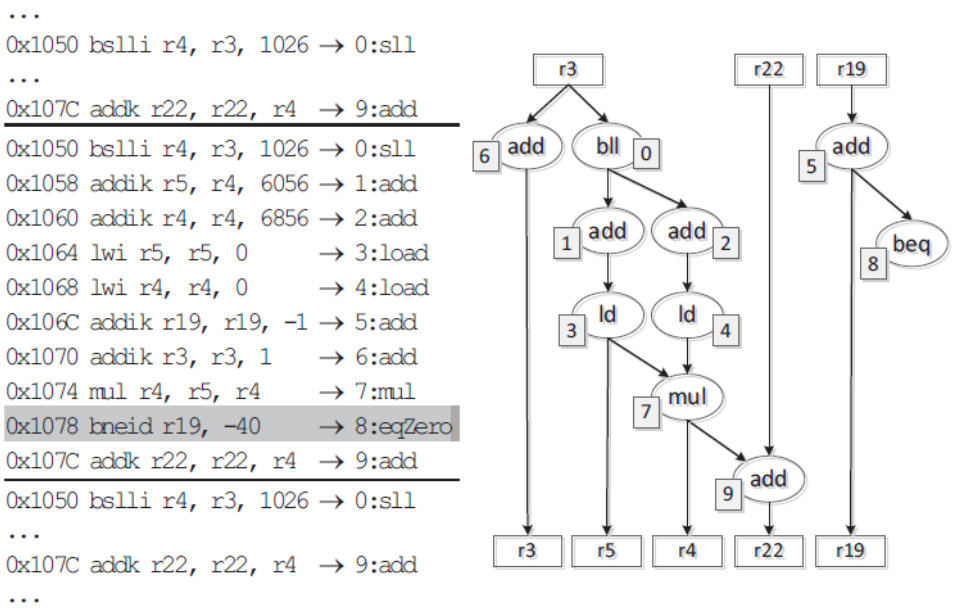

Figure 2.3: Example of a CDFG that serves as input for the scheduler and associated Microblaze instructions of the Megablock [3]

Regarding the RPU unit, [1] explores two types of architecture: 1-D and 2-D pipelined design. The second one, that is represented in Figure 2.4, is obtained "by a direct translation of the Megablock CDFGs into (pipelined) datapaths" [1]. This type of RPU can be obtained directly from the CDFG.

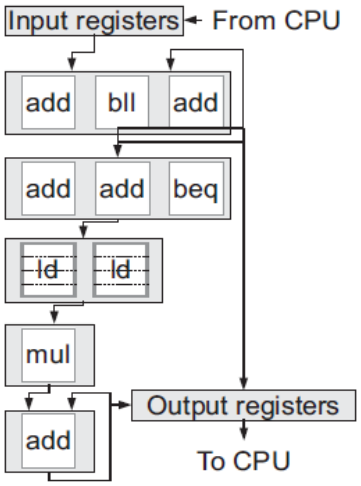

Figure 2.4: Template of the 2-D Reconfigurable Processing Unit, [1]

The 1-D RPU is the most recent implementation and explores loop pipelining via modulo scheduling. This algorithm will be discussed in the next section. Its structure is represented in Figure 2.5.

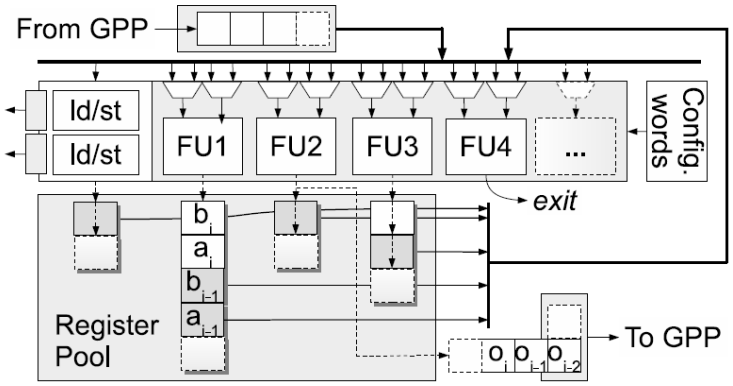

Figure 2.5: Template of the 1-D Reconfigurable Processing Unit, [1]

There are some constraints associated with the 1-D architecture of the RPU that need to be considered. Since data produced by each FU will be used in different cycles, the output registers must be able to store that information. That way, there is a FIFO register after each FU. A second problem is related to the connectivity between the output registers and the input registers of the RPU. Ideally, this connectivity should be sufficient to allow the obtained schedule; all the outputs should be connected to all the inputs, but the area utilized would grow considerably. Throughout this work, the connectivity is assumed to be enough to handle any produced schedule.

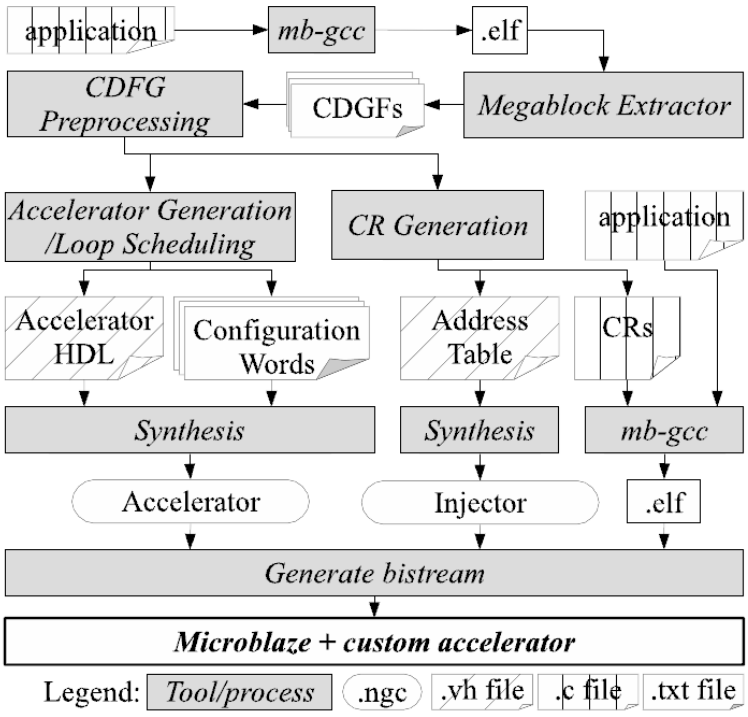

Figure 2.6: Toolflow for generation of customized loop accelerators, [4]

The scheduling algorithm used is the Modulo Scheduling technique, which will be presented in Section 2.3. The most recent implementation of this technique is done in MatLab, and it works offline. Figure 2.6 shows the steps and tasks done in [4]. After identifying the Megablock and generating the corresponding CDFG his tool is responsible, not only for generating the scheduling of instructions, but also for the generation of the custom accelerator and the required interconnections. All of this is made offline.

There are some alternative systems that are briefly described next.

2.1.1 WARP

One alternative system is the WARP processor presented in [5]. It detects frequent basic blocks. In order to do this, it does the profiling of runtime traces. The WARP processor augments a MicroBlaze with a loosely coupled custom reconfigurable fabric. It also runs a second MicroBlaze processor responsible to synthesize a circuit for each detected loop. It targets the most frequent innermost loops. The binary needs to be modified in order to migrate execution to the generated circuit. Only integer and arithmetic logic is supported.

2.1.2 ADEXOR

The ADEXOR system presented in [6] focuses on instruction set extension. There is a reconfigurable functional unit tightly coupled to the MIPS processor pipeline. It detects single-entry

multiple-exit traces with multiple paths. Profiling is made offline through simulation, and, like Warp, requires the modification of the binary. Only integer and arithmetic logic is supported.

2.1.3 ASTRO

The ASTRO system presented in [7] maximizes memory access parallelism on instruction set extension. The detection of atomic sequences of basic blocks is made offline via simulation. The synthesis of accelerators, which is also offline, is done through a direct translation of loop's CDFG. Only integer and arithmetic logic is supported.

2.2 Instruction Scheduling

"It is well known that, as a rule, there is inadequate instruction-level parallelism (ILP) between the operations in a single basic block and that higher levels of parallelism can only result from exploiting the ILP between successive basic blocks." [8].

Instruction scheduling techniques focus on improving instruction-level parallelism, in order to achieve higher performances. This is done by executing operations of different phases of execution at the same time.

There are two main groups of instruction scheduling algorithms: cyclic and acyclic schedulers. While cyclic schedulers work with loops in the program, acyclic schedulers operate on loop-free regions, [9].

2.2.1 Acyclic Schedulers

Acyclic schedulers move operations from their original basic block to another preceding/succeeding basic block. Consequently, it is necessary to have a bigger control of the data being used by each basic block, since there can be instructions being executed out of order. These type of algorithms differ in the region that is considered for the scheduler. Trace scheduling, [10] and superblock scheduling, [11] are two examples of acyclic scheduling algorithms. The main difference between them is the region that they use to schedule. In the first one, the region is a trace, which is a linear path of the code, that can have multiple entrances and exits. In superblock scheduling, the region has only one single entry. There are other types of regions, like hyperblocks and treeregions, [9].

Trace Scheduling exploits parallelism by generating code for more than one basic block at a time [12]. The traces are paths of many basic blocks. The most frequent ones are picked first, and are ordered this way. This can be seen in Figure 2.7.

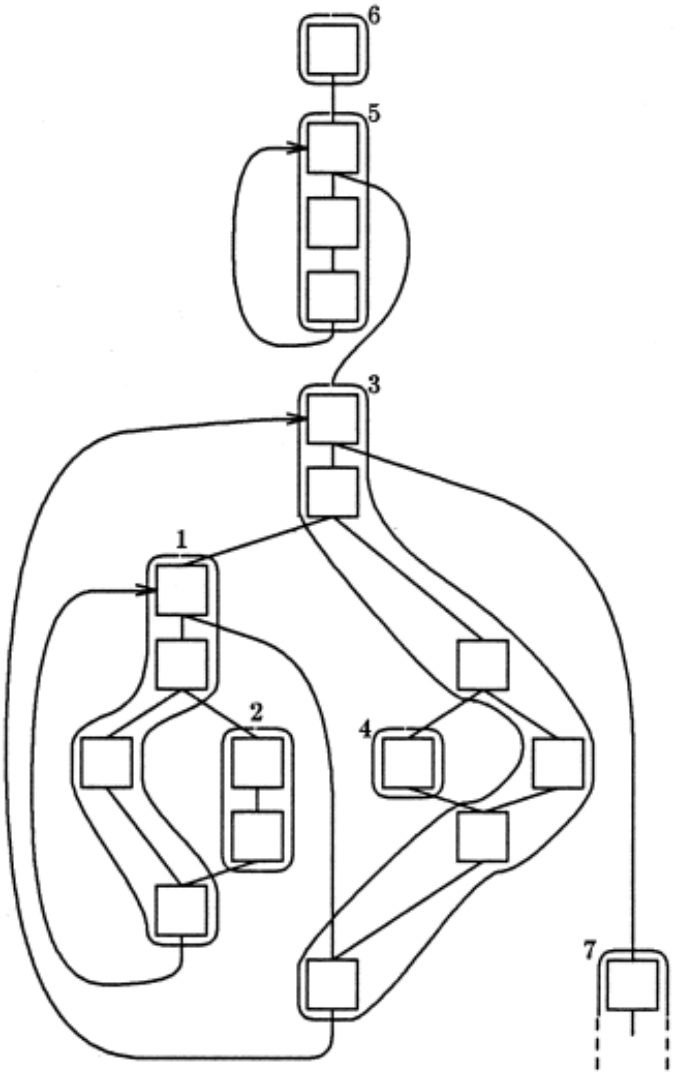

Figure 2.7: Flow graph of basic blocks, including the selection by frequency, [12]

However, this reordering of operations introduces some problems regarding data-precedence and jumps off and into the trace. In [12], a bookkeeping process is presented, which tackles issues provocated by code motion with the placement of duplicates of some operations. [13] also explores a similar approach, with the addition of Compensation Code.

Superblock Scheduling is presented in [11] as a solution for the bookkeeping complexity of trace scheduling. In this technique, the region utilized - the superblock - is similar to a trace, but has no side entrances. To achieve this, a process called tail duplication is responsible to remove those entrances, by creating a copy of the trace and moving all side entrances to the corresponding duplicate [11][14].

Both techniques also approach some code transformations like loop unrolling. This consists in replicating the body of the loop any number of times. The following example illustrates this scenario.

```
i := 0
while(1){
    if (i < n)
        break;
    //...
    //some code here
    //...
    i++;
}
```

Figure 2.8: Example code before loop unrolling

The code presented in Figure 2.8 is unrolled three times, generating the new code:

```
i := 0
while(1){
    if (i < n)
        break;
    //...
    //some code here
    //...
    i++;
    if (i < n)
        break;
    //...
    //some code here
    //...
    i++;
    if (i < n)
        break;
    //...
    //some code here
    //...
    i++;
    }
```

Figure 2.9: Example code after loop unrolling

The unrolling mechanism allows a bigger parallelism when doing the scheduling, since it will produce bigger traces or superblocks.

2.2.2 Cyclic Schedulers

On the other hand, cyclic schedulers work with loops, "moving operations across iteration boundaries", [8]. This method allows the achievement of higher levels of ILP. As discussed in [8], there are many different approaches to cyclic scheduling. Some of them unroll the loop, and then apply one of the acyclic scheduling algorithms already mentioned before. The biggest drawback is related with performance degradation at the back-edge.

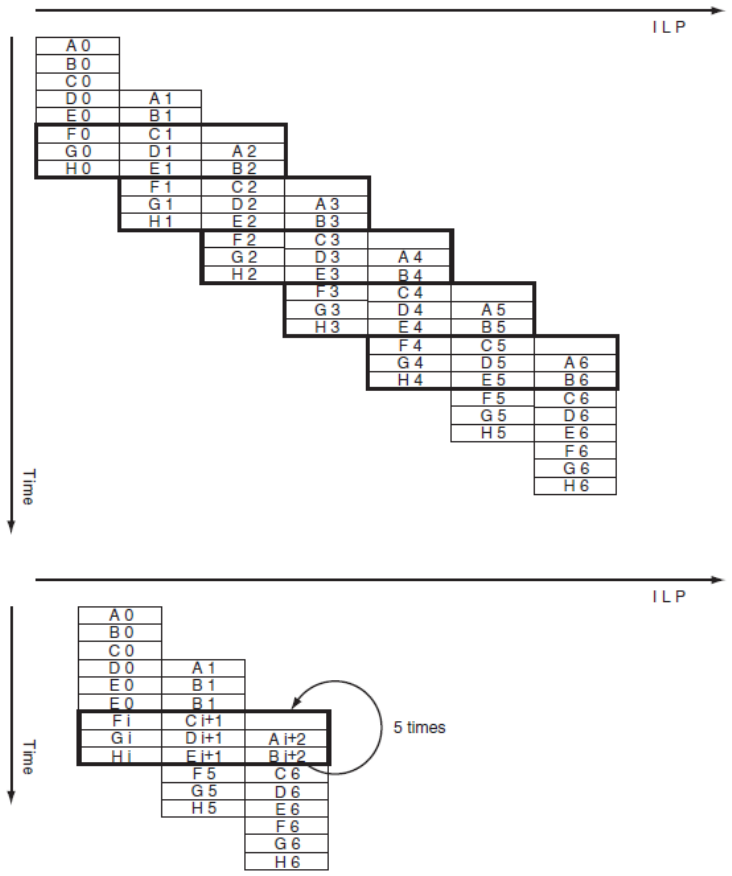

Figure 2.10: Illustration of software pipelining, [9]

Software pipelining is a different approach to cyclic scheduling. "Software pipelining is the class of global cyclic scheduling algorithms that exploits interiteration ILP while handling the back-edge barrier", [9]. As can be seen in Figure 2.10, the main objective of this type of algorithms is to find a kernel of code that can be executed repeatedly and overlaps multiple iterations. The prologue and epilogue are the sections of code before and after the kernel. Again, there is more than one approach to performing software pipelining. The “move-then-schedule” approach moves instructions across the back-edge of the loop, but there is no deterministic method to select those instructions. The “schedule-then-move” approach focuses directly on the generation of the schedule, and moves the instructions according to it. One algorithm that follows this approach is Modulo Scheduling, which will be explored in the next section.

2.3 Modulo Scheduling

Modulo Scheduling is an example of software pipelining. It follows a "schedule-then-move" approach [8]. This means that the algorithm's focus is the maximization of performance. In order to achieve this, there is a simultaneous unrolling and scheduling of the loop in study, until one gets a schedule kernel that can be repeated. The greatest advantage of this technique is that it allows overlapping different iterations of a loop [15]. Every iteration of the loop must be scheduled in the same way, in order to avoid different code for each iteration.

The first part of Modulo Scheduling is the calculation of the Minimum Iteration Interval (MII). The Iteration Interval (II) determines the interval at which new iterations can start. The MII can be determined, according to [8], through the calculation of the resource-constrained II (ResII) and the recurrence-constrained II (RecII). MII is the maximum value between them. ResII reflects the influence of the available hardware in the obtained schedule. If there are few units capable of doing a certain instruction which will be executed many times in a loop, the II needs to increase in order to allow the execution of all operations. RecII is calculated through the analysis of chains of dependences in a Megablock. If there are sucessive instructions depending on data provided by previous operations, the II may need to increase in order to avoid precedences violations.

The scheduling algorithm is quite simple, which is a big advantage of this technique. The operations are allocated one at a time (there is a priority associated to each instruction to select which one to schedule next) to a certain time-slot. This can be represented in a table, the Modulo Reservation Table (MRT), as represented in Figure 2.11.

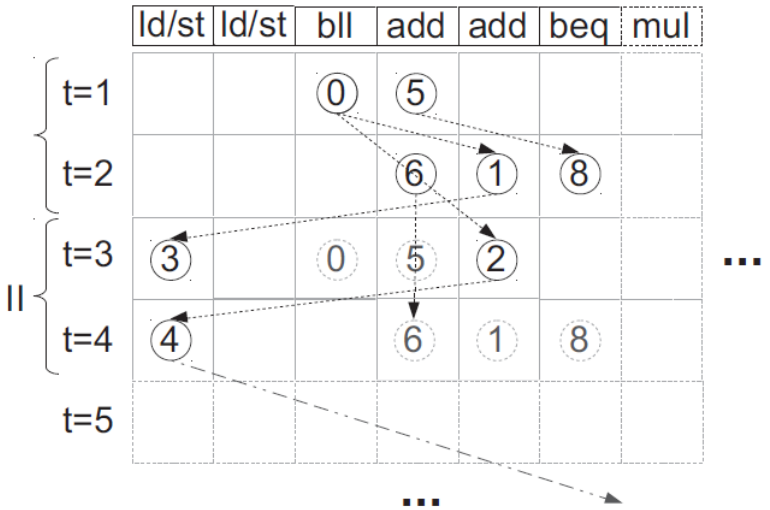

Figure 2.11: Table that serves as an example of Modulo Scheduling, [3]

One of the first works to propose this technique is [16]. It was developed in order to schedule inner loops in a CGRA tightly coupled to a VLIW processor. There are other approaches that are targeted to CGRA like [17] which prioritizes routing and performs placement and scheduling during routing process [18] which focuses on minimizing the compilation time.

The base for the implementation of this technique done in this work is the Iterative Modulo Scheduling presented in [8] and will be further explained in Section 4.2

Chapter 3

Problem Description and General Development Approach

This chapter presents a more detailed description of the problem in analysis. It also describes the approach that is going to be followed throughout the development of the solution.

3.1 Problem Description

As explained in Chapter 2, the system of [1] has four main steps [1]:

1. Megablocks are identified from execution traces of the application, obtained from an instruction set simulator;
2. Selected Megablocks are translated to a RPU specification and corresponding configurations, producing a customized accelerator and all data necessary to configure the system, without modifying the software binaries;
3. At runtime, the GPP is monitored to detect the imminent execution of the regions of code translated to hardware;
4. Execution is then migrated transparently to the RPU.

The objective of such a project is to create a fully dynamic system, however, some of the parts are still being processed offline. This is the case with two of its major blocks: the trace recognition tool and the scheduler. In this project, the main focus of study is the implementation of an embedded scheduler.

The embedded scheduler should be responsible for the following tasks:

- Gather information of the Megablock identified
- Generate the scheduling of the Megablock identified

3.1.1 Megablock Identification

The trace identification mechanism will be the same used by Nuno Paulino [1]. The Megablock Extractor tool [19] identifies the Megablocks simulating a cycle-acurate Microblaze using the binary file (ELF format) and produces some files containing information about the CDFG produced for each Megablock. Examples of information of these files are presented in Figures 3.1 and 3.2.

```
// Liveouts
numLiveouts:3
opNumber:0
outputIndex:0
liveoutReg:r5
opNumber:2
outputIndex:0
liveoutReg:r29
opNumber:0
outputIndex:1
liveoutReg:rmsr_29
```

Figure 3.1: Part of an output file with information about the Megablock

```
OP:0
addr:0x00002DB8
operation:add
level:1
numInputs:2
inputType:livein
inputValue:r5
inputType:livein
inputValue:r5
numUsedOutputs:1
outputId:0
outputFanout:1
```

Figure 3.2: Part of an output file with information about an operation

The extractor tool also produces files containing the assembly instructions of the Megablock in study, as can be seen in Figure 3.3.

```
0x00002DB8 add r5, r5, r5      -> 0:add
0x00002DBC bgtid r5, -4        -> 1:lessOrEqualZero
0x00002DC0 addik r29, r29, -1 -> 2:add
```

Figure 3.3: Output file with the assembly instructions of the Megablock

3.1.2 Constraints

The algorithm that is going to be used to generate the schedule is the modulo scheduling technique explained in Chapter 2. The Iteration Interval (II) is defined by the characteristics of the identified Megablock.

Additionally, it is necessary to make some assumptions before performing the scheduling:

- There is only one type of RPU available,
- The RPU has a well defined architecture,
- The connectivity will not limit the obtained schedules,
- The output FIFO's have sufficient size to handle the obtained schedules.

3.2 Validation Mechanisms

Throughout the development of the implementation, it was important to check the correctness of work done. With this in mind, both Nuno Paulino's approach of Modulo Scheduling in MatLab [1] and the Megablock Extractor tool [19] were used to confirm the CDFG informations, the schedule produced and the configuration word generated. Additionally, the schedules were manually verified to check for any type of violation, such as precedences, functional units attribution to operations or non scheduled instructions.

Since the configurations words generated should be the same as in the previous work, there was no need to verify if the system provides the same results as the non-accelerated version, since this was already proven in [3].

3.3 Experimental Setup

The development and analysis of the implementation will have two different stages. Initially, a simulation platform was utilized since custom hardware design for FPGA's is a lengthy process and a simpler and faster testing environment would speed up the work progression.

OVPSim [20] is an open source simulation environment that permits to simulate platforms and processors. That being said, it was an useful tool to simulate a system with Microblaze processor. Not only does it permit a fast simulation of the system, but it also provides information regarding the approximate execution time in a real system. This was useful to build a base of comparison for further measurements.

After the validation of the implementation, tests were made in a Xilinx VC707 board, which is equipped with a xc7vx485 Virtex-7 FPGA. Vivado Design Suite [21] was used to generate the bitstream. A simple system with a Microblaze processor and a timer (to enable the measure of execution times) was designed. This system is presented in Figure 3.4. In all the tests, the synthesis effort policy was set to speed, and the placement and routing effort was set to high.

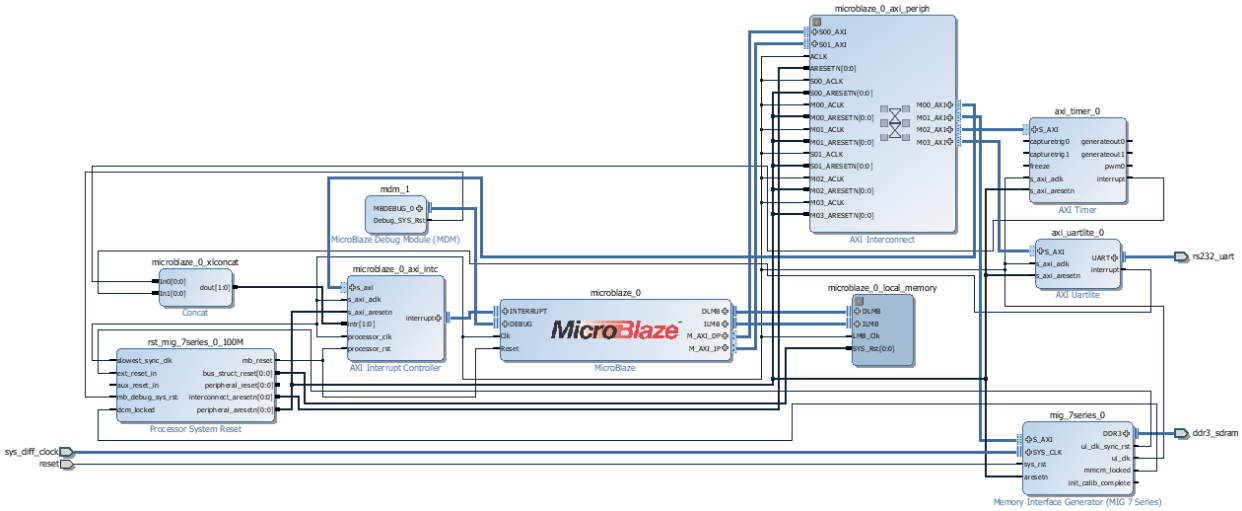

Figure 3.4: Microblaze based system implemented in the FPGA

Chapter 4

Algorithm Description and Implementation

This chapter presents a more detailed description of the Iterative Modulo Scheduling algorithm and the implementation developed. It also describes the organization of structures containing the information of Megablocks.

4.1 Megablocks and Graphs Information

All the information gathered about the identified repetitive sequences of instructions - Megablocks - follows the organization described in [22]. The assembly instructions of the MB are analyzed and the corresponding CDFG is constructed. The nodes and connections represent the relationship between data and operations and some additional information such as exit points. The are four types of nodes:

- Operation,
- Livein,
- Constant,
- Exit.

An Operation node represents an instruction of a Megablock, like an add or a mul. The Livein node is related with external values which are obtained before the execution of the loop. The Constant node represents a literal value. Finally, the Exit node is an exit point of the Megablock.

Regarding the connections, there are five different types:

- Data,
- Control,
- Liveout,

- Feedback,
- ExitAddress.

The Data connections cointain information about the flow of data between outputs and inputs of operations. This way, only Operation, Constant and Livein nodes can have Data connections.

The Control connections' purpose is to indicate if an Exit point is triggered. These connections are always between Operation and Exit nodes.

The Liveout connections represent the value of outpus of the Megablock. They always connect to Exit nodes and can be originated by Operation, Constant or Livein nodes.

The Feedback connection is related with internal updates to the values of Livein nodes.

Finally, the ExitAddress connections indicates the instruction address to resume execution after the accelerated Megablock. It always connects to an Exit node.

The following subsections will present the organization of information related with CDFGs in the implementation developed throughout this work.

4.1.1 Megablock Information

One of the utilized structures is the Megablock struct, created by [1] which contains general information about the identified loop. The most relevant variables of this struct can be seen in Figure 4.1.

```
struct megablockGraph {
    /// General info
    int startPC;
    int depth, width;
    /// Inputs and outputs
    int  numliins, numliouts;
    struct liveout *liveouts;
(...)
    /// Operations
    struct operation *operations;
    int numops,
        numsts,
        numlds,
        numexits,
        totaloutputs;
    };
```

Figure 4.1: Some of the variables of the Megablock structure

The depth and width values indicate the dimensions of the CDFG. The variable startPC contains the address of the first instruction of the accelerated loop. Regarding inputs and outputs,

numliins and numliouts represent, respectively, the number of liveins and liveouts of the graph. The struct liveout contains information of all the liveouts of the graph, such as the CPU register to be updated and the source of the value. Struct operation will be explained in the next subsection. The remaining variables contain information about the total number of operations, the number of store and load instructions, the number of exit points and the total number of outputs.

4.1.2 Operation Information

In 4.2 it is presented the composition of the Struct Operation.

```
struct operation {

    int op_num,
        op_addr,
        op_class,
        op_level,
        op_ASAP,
        op_ALAP,
        op_numinputs,
        op_numoutputs,

    struct input *op_inputs;
    struct output *op_outputs;
    };
```

Figure 4.2: Operation structure

This structure contains information about the number of the operation (operations are numbered by the order of the assembly instructions) and the address of the instruction.

The level is related with the position of each operation on the CDFG. ASAP and ALAP mean "as soon as possible" and "as late as possible", respectively, and are essential to define the time slots where each operation can be fitted.

The final two int variables stand for number of inputs and outputs. Both structures contain relevant information about inputs and outputs and will be described in Subsections 4.1.3 and 4.1.4.

4.1.3 Inputs Information

In Figure 4.3 it is presented the composition of the Struct Input.

```
struct input {

    struct operation *parent;
    enum inputType input_type;
    int input_value[2];
    int firstfeed;

    };
```

Figure 4.3: Input structure

The parent is the operation who owns this input. From the input_type value it is possible to know if the input is a constant or an output of another instruction or even a value from a register. The input_value variable is essential in the scheduling phase, since it gives information about the origin of a certain input. If it comes from another instruction, it will contain the information about which operation and from which output the value can be obtained.

4.1.4 Outputs Information

Figure 4.4 presents the composition of Struct Output.

```
struct output {

    struct operation *parent;
    int output_index;
    int output_fanout;

    };
```

Figure 4.4: Output structure

Like in the input structure, the parent is the operation who owns this output. The remaining variables indicate which of the outputs is utilized and to how many inputs it connects.

4.2 Iterative Modulo Scheduling

In this section, the Iterative Modulo Schedule algorithm proposed in [8] is explained.

Modulo Scheduling requires the definition of an Iteration Interval (II) to begin the scheduling. [8] defines two methods of defining a minimum II: resource-constrained and recurrence-constrained. In [1], and since the RPU is customizable, the first one was not critical. But in this work, the RPUs are defined a priori. This way, both methods will affect the minimum II calculation for the run-time implementation. This minimum II value is a lower bound for the II. That

way, it may not be possible to have a schedule with that II, but it is guaranteed that there is no lower II which permits a valid schedule.

Afterwards, the iterative algorithm is started with the minII value. The choice of the operation to schedule is made through a height-based priority function which tries to tackle the difficulties of scheduling strongly connected component (paths were every vertex has dependences to every other vertex). The equation for the priority calculation is presented in equation 4.1.

$$HeightR(P) = \begin{cases} 0, if\ P\ is\ the\ STOP\ pseudo-op \\ Max(HeighR(Q) + Delay(P,Q) - II * Distance(P,Q)), otherwise \end{cases} \tag{4.1}$$

The next step is to calculate the minimum and maximum times to place the operation. The minimum time is calculated through the Earliest Time equation presented in equation 4.2.

$$Estart(P) = \begin{cases} 0, if\ q\ unscheduled \\ Max(0, SchedTime(Q) + Delay(P,Q) - II * Distance(P,Q)), otherwise \end{cases} \tag{4.2}$$

It is not necessary to search for time slots more than II contiguous from the minimum time. If it is impossible to schedule in this interval, there will not exist a slot outside this range to schedule the operation. This way, the maximum time calculation is done with the following formula:

$$MaxTime = Estart + II - 1 \tag{4.3}$$

The following step is done by calling the FindTimeSlot function which is responsible to find a suitable time slot for the operation. All time slots between the minimum and maximum times are checked, and the first empty one is chosen. If there is no valid situation, then it should be done one of the following options (the first has higher priority):

1. Choose the minimum time as the time slot, either if it is the first time that the operation is being scheduled, or if the minimum time is greater than the time at which the operation was previously scheduled,

2. Choose the time at which the operation was previously scheduled incremented by one.

The scheduling of an operation only respects precedence relation with previous operation through the calculation of the Earliest Start Time. In order to avoid violating any precedence relation, all operations which are successors of the latest scheduled operation are unscheduled.

There is also a Budget variable in [8], which is decremented every iteration. If it reaches 0, the scheduler is restarted with a greater II.

4.3 Implementation

In this section, the implementation of the Iterative Modulo Schedule algorithm, which is represented in Figure 4.5, is described.

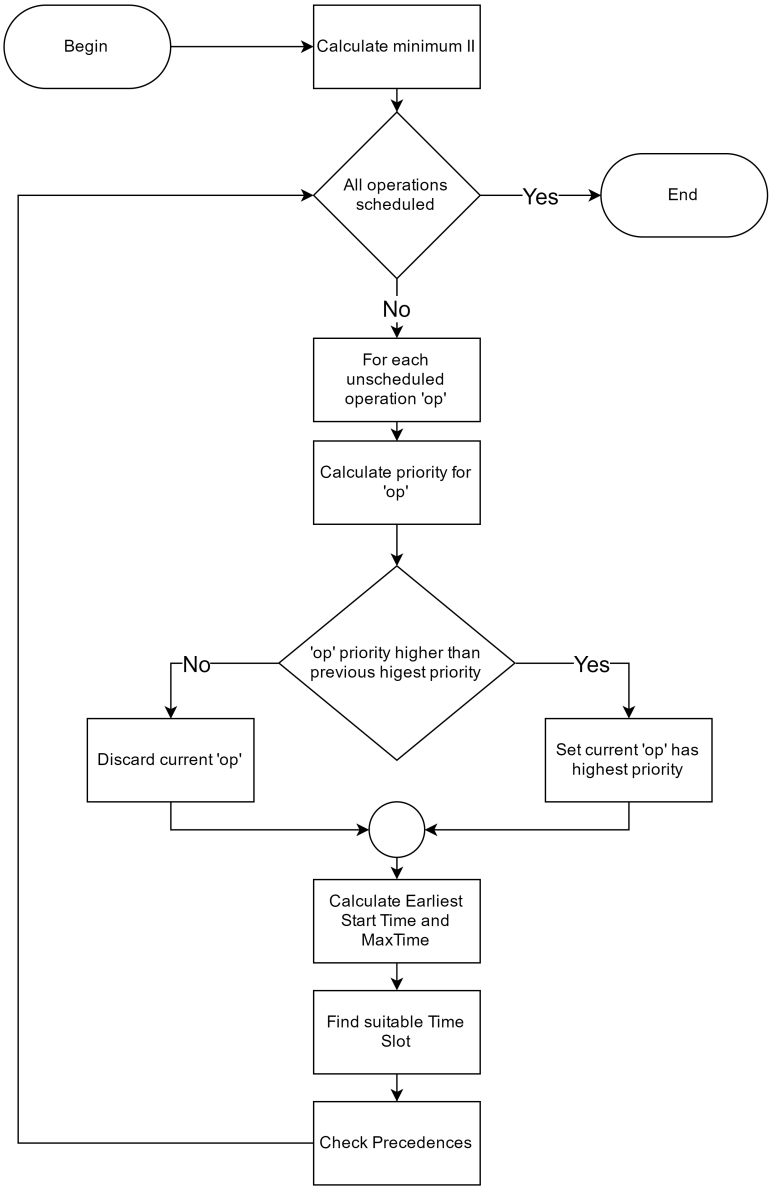

Figure 4.5: Diagram of the Iterative Modulo Schedule algorithm implementation

4.3.1 Minimum Iteration Interval

In order to determine the Iteration Interval, it was necessary to define how to calculate the Resource II and the Recurrence II.

The Resource Iteration Interval is quite simple to determine, since it reflects the usage requirements imposed by one iteration of the identified loop. As can be seen in equation 4.4, the ResII

results from the relation between the number of operations that require a certain Functional Unit, and the effective number of available FU's in the RPU.

$$ResII = max(\left\lceil \frac{\sharp operations}{\sharp FU_i} \right\rceil) \tag{4.4}$$

In order to achieve this, it is necessary to associate each of the Megablock's instructions to one type of the available FU's. This being done, the number of operations associated with each FU is counted and the ResII is calculated.

The Recurrence Iteration Interval is obtained from existent dependences on the CDFG. Instead of searching for the chain of dependencies for all operations, the op_level value, which can be obtained from the output structure, gives information about the number of operations that precede a certain one. This way, the RecII will be the addition of the level of precedent operations - those which are the source for the inputs - and the associated delay. The RecII will be the maximum value obtained.

As can be seen in equation 4.5, the RecII

$$RecII = max(operation_i\_level + operation_i\_delay + 1) \tag{4.5}$$

Finally, the Minimum Iteration Interval value will be the maximum between the RecII and the ResII.

4.3.2 Priorities

The Iterative Modulo Schedule presented in [8] uses a height-based priority function. Some other approaches also try to give higher priorities to instructions which are on the end of a chain of dependencies. In this work, a simpler approach was made by giving higher priorities to operations which "as late as possible" value is lower.

Additionally, since operations can be unscheduled and rescheduled throughout the iterations of the algorithm, a higher priority is given to those operation who have been scheduled before. This can be seen in equation 4.6

$$Priority = t_{ALAP}/scheduled; \tag{4.6}$$

The operations which have lower values of priority will be scheduled first. The scheduled value is two for operations that were already scheduled before and one if it is the first time the operation is going to be scheduled.

4.3.3 Earliest Start Time

The Earliest Start Time calculation has a similar approach to the calculation of the RecII and it follows the presented in equation 4.7

$$Estart = t_{input} + delay_{input} - II * distance(input) \tag{4.7}$$

The Earliest Start Time results from the addition of the scheduled time of a source with the delay associated with that operation. Since a source operation can be an instruction from a previous iteration of the cycle, it is possible to subtract the II to have an Earliest Start Time which represents the first timeslot of the MRT where the operation can be scheduled - the kernel will have operations from one iteration executing during or before the end of the previous iteration. This is the same approach of [8], which presents the same equation.

4.3.4 Time Slots

In order to find a suitable Time Slot for an operation, it is necessary to calculate the minimum and maximum times where it can be scheduled. The minimum time is the Earliest Start time of the operation. The maximum time results from equation 4.8.

$$maxTime = minTime + II - 1 \tag{4.8}$$

This is the same approach made in [8]: "It is pointless and redundant to consider more than II contiguous time slots starting with Estart. If a legal time slot is not found in this range because of resource conflicts, it will not be found outside this range. Therefore, maxTime is set equal to Estart + H -1."

4.3.5 Validation of Precedences

One problem related with the Iterative Modulo Schedule algorithm is that the possibility of rescheduling operation may arise problems with operations' precedences. In order to check if there is any precedence violation, the earliest start time is calculated for every scheduled instruction and compared with the timeslot where they were placed.

4.3.6 Schedule

Before attempting to schedule any instruction, the MRT is empty and all operations are marked as never scheduled before.

The first step of the iterative algorithm is the choice of an operation. This is made through the comparison of priorities of all operations, which are calculated as described in Subsection 4.3.2. Afterwards, the Earliest Start Time is determined, and the scheduler tries to find a time slot to place the operation, as explained in Subsection 4.3.3. The final step of an iteration is to verify all precedences.

The scheduler will abandon the current II after executing a predefined number of iterations in the attempt of producing a valid schedule. The II is increased, and the scheduler is restarted. Also, it is possible to define a maximum value for the II, when the application should stop trying to produce a valid schedule.

4.3.7 Prologue and Epilogue

After obtaining the kernel of Iterative Modulo Schedule, it is necessary to produce the Prologue and Epilogue parts of the schedule.

The Prologue is done by analysing, starting at the first operation, which instructions can be executed in the same II cycle. First, it is verified if it is possible to execute all operations in one II cycle, and, if not, the number of II cycles is increased until it is possible to execute all the operations of the MRT.

The Epilogue is done in a similar way. Instead of adding up more operations until all are being executed, in each II cycle, the first instructions are removed from the schedule, until there are no remaining instructions that are the source of an input of a following instruction.

Chapter 5

Validation and Results Analysis

This chapter presents the obtained results for the validation tests and some measured times of the implementation developed.

5.1 Benchmarks

In order to validate the implementation of the Iterative Modulo Schedule algorithm and perform time measurements, two sets of benchmarks were utilized: one is a set of floating-point benchmarks from [23] and the other is a set of integer benchmarks from [24].

The two sets of benchmarks have the following characteristics:

kernel	Number of Operations	Memory Instructions	Number of Exit Points
quantize	18	3	1
conv3x3	61	3	2
perimeter	28	3	2
boundary	20	3	2
sad16x16	15	1	3
mad16x16	15	1	2
sobel	35	3	2
dilate	119	1	3
erode	127	1	3

Table 5.1: Characterization of the integer set of benchmarks

kernel	Number of Operations	Memory Instructions	Number of Exit Points
cholesky	19	3	1
diffpredict	54	3	2
glinrecurrence	13	1	1
hydro	14	1	1
hydro2dimp	30	3	2
intpredict	49	3	2
linrec	56	3	2
matmul	22	3	1
statefrag	27	3	2

Table 5.2: Graph information of the floating-point set of benchmarks

5.2 Validation

In order to validate the implementation of the Iterative Modulo Schedule algorithm, four types of RPUs were utilized. RPU1, RPU2 and RPU3 have two M_MEM units, one B_UNIT unit, one S_BOR unit, one A_MUL unit and one S_SUBC unit. RPU1 has two A_ALU units, RPU2 has four A_ALU units and RPU3 has 8 A_ALU units. Additionally, a custom RPU is obtained from the Matlab implementation of Nuno Paulino [1].

For the floating-point set of benchmarks, all RPUs have an additional F_FPU unit.

In Table 5.3 the IIs obtained for integer set of benchmarks are presented and in Table 5.4 the IIs obtained for the floating-point set of benchmarks are presented

kernel	RPU1	RPU2	RPU3	Custom RPU
quantize	3	3	3	3
conv3x3	13	11	10	10
perimeter	5	4	3	3
boundary	5	5	4	4
sad16x16	3	2	2	2
mad16x16	3	2	2	2
sobel	7	7	6	5
dilate	24	22	20	20
erode	26	24	22	22

Table 5.3: II obtained for the integer set of benchmarks

kernel	RPU1	RPU2	RPU3	Custom RPU
cholesky	4	3	3	3
diffpredict	11	10	10	10
glinrecurrence	3	3	3	3
hydro	3	3	3	3
hydro2dimp	6	6	6	6
intpredict	10	10	10	10
linrec	11	11	11	11
matmul	4	3	3	3
statefrag	6	6	5	5

Table 5.4: II obtained for the floating-point set of benchmarks

The obtained results were compared with the ones obtained by [1], and all of them were correct. Regarding the schedules, there were some occasional differences regarding the placement of instructions which had more slack, but the produced schedule was equally correct.

5.3 Results Analysis

In order to measure the execution times of the developed implementation, three different platforms were utilized:

- Microblaze processor running on a Virtex 7 FPGA,
- OVPSim Microblaze environment,
- Matlab implementation.

Regarding the hardware, it was utilized the RPU2 described in the previous section, which contains two M_MEM units, four A_ALU units, one B_UNIT unit, one S_BOR unit, one A_MUL unit, one S_SUBC unit and an additional F_FPU unit for the floating-point set of benchmarks.

kernel	Microblaze time(s)	Simulation time(s)	MatLab time(s)
quantize	0,089	0,081	0,44
conv3x3	0,30	0,28	1,47
perimeter	0,12	0,083	0,40
boundary	0,15	0,11	0,59
sad16x16	0,061	0,051	0,29
mad16x16	0,084	0,063	0,27
sobel	0,14	0,14	0,74
dilate	0,41	0,55	2,95
erode	0,81	0,61	3,24

Table 5.5: Times obtained in OVPSim, in the FPGA and with the Matlab version for the integer set of benchmarks

kernel	Microblaze time(s)	Simulation time(s)	MatLab time(s)
cholesky	0,078	0,072	0,43
diffpredict	0,26	0,27	1,43
glinrecurrence	0,077	0,086	0,51
hydro	0,69	0,66	0.40
hydro2dimp	0,16	0,14	0,86
intpredict	0,31	0,33	1,47
linrec	0,29	0,26	1,57
matmul	0,081	0,080	0,43
statefrag	0,13	0,12	0,71

Table 5.6: Times obtained in OVPSim, in the FPGA and with the Matlab version for the floating-point set of benchmarks

Analysing the times presented in Table 5.5 and in Table 5.6 it is possible to understand that there is an overall speedup from the MatLab implementation. This was expected since the MatLab version was not developed to be fast and it is also responsible for the generation of the accelerator's HDL.

The times of both FPGA and simulation measurements are very similar, and they show that this algorithm does not consume a lot of time to produce schedules.

Chapter 6

Conclusions and Future Work

This chapter presents some conclusions about the work developed and some possible future modifications of the implemented algorithm.

6.1 Conclusions

The proposed objectives for this work were achieved. The chosen scheduling technique - the Iterative Modulo Schedule - was successfully implemented and adapted to the available system. During the dissertation, the correctness of the system was proven, and it was also achieved another step in the direction of creating an autonomous and self accelerating system.

Finally, there were obtained a number of results which allowed to prove the correctness of the implementation, and, also, to demonstrate the low execution time of the implementation. Additionally, it is important to understand that the main goal of a fully autonomous system is to accelerate instructions on a long time run. This way, and since the scheduling is only run once, at the beginning of execution, its impact is diluted with the increasing time of execution of the system.

6.2 Future Work

The current implementation has some constraints that were described in Subsection 3.1.2. In order to achieve a system with better performance and a smaller area usage, the connectivity should be allowed to be limited, forcing the scheduler to be "intelligent" in order to check the dependencies of operations and the existence of such interconnection. Additionally, the output FIFO length should be reduced as much as possible, which would affect the selection of time slots by the schedulers, since some outputs may be lost after some cycles.

Regarding the RPUs, there are some improvements than can be made. Instead of assuming the existence of only one well defined RPU, there can exist a small set of predefined RPU's, and the scheduler should be able to select the most best-suited one for each Megablock in execution. A small step in this direction was made, since the scheduler is capable of comparing the minII

associated to a number of RPUs. However, it is necessary to take into account the time wasted on the reconfiguration of the interconnections with each RPU, and the impact of the increase of utilized area. Another possible approach is to have a small RPU, and customize it with the addition of FU's according to the Megablock in execution. Again, the scheduler must be able to evaluate the time spent on the reconfiguration of the system and, also, determine which FUs should be added to the existing RPU.

References

[1] Nuno Paulino, João Canas Ferreira, João Bispo, and João M. P. Cardoso. Transparent acceleration of program execution using reconfigurable hardware. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 1066–1071, San Jose, CA, USA, 2015. EDA Consortium. URL: http://dl.acm.org/citation.cfm?id=2757012.2757061.

[2] J. Bispo and J.M.P. Cardoso. On identifying segments of traces for dynamic compilation. In *2010 International Conference on Field Programmable Logic and Applications (FPL)*, pages 263–266, Aug 2010. doi:10.1109/FPL.2010.61.

[3] J. Bispo, N. Paulino, J.M.P. Cardoso, and J.C. Ferreira. Transparent trace-based binary acceleration for reconfigurable hw/sw systems. *IEEE Transactions on Industrial Informatics*, 9(3):1625–1634, Aug 2013. doi:10.1109/TII.2012.2235844.

[4] Nuno Paulino. Transparent hardware generation from assembly code at program execution time, 2016. URL: https://paginas.fe.up.pt/~dee11006/doku.php?id=implementations#toolflow.

[5] Roman Lysecky and Frank Vahid. Design and implementation of a microblaze-based warp processor. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(3):22, 2009.

[6] Hamid Noori, Farhad Mehdipour, Koji Inoue, and Kazuaki Murakami. Improving performance and energy efficiency of embedded processors via post-fabrication instruction set customization. *The Journal of Supercomputing*, 60(2):196–222, 2012.

[7] Mingjie Lin, Shaoyi Chen, Ronald F DeMara, and John Wawrzynek. Astro: Synthesizing application-specific reconfigurable hardware traces to exploit memory-level parallelism. *Microprocessors and Microsystems*, 39(7):553–564, 2015.

[8] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM. URL: http://doi.acm.org/10.1145/192724.192731, doi:10.1145/192724.192731.

[9] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. Embedded computing. San Francisco, 2005. Morgan Kaufmann.

[10] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981. doi:10.1109/TC.1981.1675827.

[11] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E.

Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *THE JOURNAL OF SUPERCOMPUTING*, 7:229–248, 1993.

[12] John Rolfe Ellis. Bulldog: A compiler for VLIW architectures. Technical report, Yale Univ., New Haven, CT (USA), 1985.

[13] P Geoffrey Lowney, Stefan M Freudenberger, Thomas J Karzes, WD Lichtenstein, Robert P Nix, John S O'donnell, and John C Ruttenberg. The multiflow trace scheduling compiler. *The journal of Supercomputing*, 7(1-2):51–142, 1993.

[14] William Y Chen, Pohua P. Chang, Thomas M Conte, and Wen-mei W. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Transactions on Computers*, 42(9):1045–1057, 1993.

[15] B Ramakrishna Rau and Christopher D Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *ACM SIGMICRO Newsletter*, volume 12, pages 183–198. IEEE Press, 1981.

[16] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings-Computers and Digital Techniques*, 150(5):255–61, 2003.

[17] Hyunchul Park, Kevin Fan, Scott A Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176. ACM, 2008.

[18] Ricardo Ferreira, Vinicius Duarte, Waldir Meireles, Monica Pereira, Luigi Carro, and Stephan Wong. A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 188–195. IEEE, 2013.

[19] João Bispo. Megablock extractor, 2013. URL: https://sites.google.com/site/specsfeup/resources/tools#TOC-Megablock-Extractor.

[20] Imperas Software. Open virtual platforms, 2016. URL: http://www.ovpworld.org/.

[21] Xilinx. Vivado design suite, 2016. URL: http://www.xilinx.com/products/design-tools/vivado.html.

[22] João Carlos Viegas Martins Bispo. *Mapping Runtime-Detected Loops from Microprocessors to Reconfigurable Processing Units*. PhD thesis, Instituto Superior Técnico, 2012.

[23] Tim Peters. Livermore loops coded in c, 1992. URL: http://www.netlib.org/benchmark/livermorec.

[24] Texas Instruments. Tms320c6000 image library (imglib) - sprc264, 2008. URL: http://www.ti.com/tool/sprc264.